Pascal–2 V2.1/RT-11 Debugger Guide

Debugging tools help uncover "run-time" errors—errors in a program's execution—that cannot be caught during compilation. For example, a procedure may generate an incorrect number of loops or make a legal but unintended change in the value of a variable. The Pascal–2 Debugger lets you control a program's execution interactively; you may suspend execution at particular statements to examine or modify the values of variables, or you may execute statements one at a time to trace the actions leading to an incorrect result.

When called, the Pascal–2 Debugger keeps track of all constants, variables, local procedures and functions and all standard and user-defined data types. The Debugger can show what's happening to data and allow you to change the data as the program executes. You can display the original source text of your program for immediate identification of context, and you can access and debug external procedures and functions called by the main program. (See "Debugging External Modules" at the end of this guide for details.) The Debugger also traps errors by halting execution of a program at the point of breakdown and identifying the last statement executed. Taken together, these features allow you to trouble-shoot a program until you have detected and corrected any errors.

This guide serves as an introduction to the Pascal–2 debugging process and as a comprehensive resource for operation of the Pascal–2 Debugger. The guide provides:

- An overview of the Pascal–2 debugging environment.
- Detailed descriptions of the Debugger commands.
- A tutorial that demonstrates the context in which Pascal–2 Debugger commands are most frequently used.
- An explanation of how external modules are debugged.
- A one-page summary of Debugger commands.

A word of warning before beginning: specifying the debugging option causes the compiler to include a call to the Debugger before each procedure and statement, which substantially increases the size of your program. The object module created by the compiler contains extra code to locate statements and procedures in your program. Moreover, introducing the Debugger turns off optimizations that would interfere with debugging. The compiler normally folds similar statements into one section of code and optimizes the usage of some variables by keeping their values in registers on the stack temporarily. These optimizations would keep the Debugger from setting breakpoints in statements and from changing the values of variables while your program was running—both of which are important debugging facilities. A little bit of memory is saved during use of the Debugger by disabling the procedure walkback—this happens automatically when the Debugger is implemented—but in general, the overhead involved in using the Pascal–2 Debugger is about one word per Pascal statement and about six words per procedure. Code returns to its normal size once you correct any problems and recompile your program without a call to the Debugger. (See "Overlays" in this guide regarding what to do if the program grows too large.)

Including the Pascal–2 Debugger in Your Program

The **debug** compilation switch invokes the Pascal–2 Debugger. (See the User Guide for details on compilation switches.) Using the **debug** switch in your compilation command automatically generates a formatted listing file, with an .LST extension, in the same directory as the output file. The Debugger reads this listing file to display the source lines when statements are identified. The Debugger can use only the listing file produced by the **debug** switch.

The **debug** switch also causes the compiler to create symbol table and statement map files in the same directory as the output file. The symbol table file (extension .SYM) describes the constants, types, variables, and the memory layout of variables. The symbol table file also contains information about each procedure and function local to the compilation unit. The statement map file (extension .SMP) contains a map of the location of the statements and their position in the listing. Both files are in binary form and are not readily examined by users.

```
.R PASCAL
*ROTAT/DEBUG

.LINK ROTAT,SY:PASCAL
```

Identifying Pascal Statements

Remember, the **debug** switch automatically generates a listing file. As the example ROTAT.LST shows, a listing file has two columns of numbers. The leftmost column lists the line numbers in the source file. The second column contains the number of each statement in the program, beginning with '1' for each procedure or function. These numbers identify points where you may set breakpoints to interrupt program execution. In ROTAT.LST, several lines accessible to the Debugger have been labeled by procedure name and statement number. As shown, statements in the main body of the program are considered to be in the procedure **MAIN**. All Pascal programs begin executing at **MAIN**,1.

You should have a printout of the listing file as reference when you begin a debugging session, or you may use the **L** command to list parts of the program while you are debugging.

Example:

```
Pascal-2 RT11 SJ V2.1D   9-Feb-84   7:06 AM     Site $1-1     Page 1-1
Oregon Software, 6915 SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202
ROTAT/DEBUG
 Line Stmt
    1       program Rotat; { rotate an array of numbers }
    2
    3         const Arraylen = 7;
    4
    5         type  Index = 1..Arraylen;  Element = 0..10;
    6               Numbers = array [index] of Element;
    7
    8         var   I: Index;  N: Numbers;  Left, Right: Index;
    9
   10         procedure Rotate(First, Last: Index;
   11                          var A: Numbers);
   12           var  I: Index;
   13
   14     1     begin
   15     2       for I := First to Last do
   16     3         A[I] := A[I + 1];     ---------- ROTATE, 3
   17     4       A[Last] := A[First];    ---------- ROTATE, 4
   18     5       write('Rotated ', first: 1, ' thru ', last: 1, '=');
   19           end;
   20
   21     1   begin { main program }     ---------- MAIN, 1
   22     2     for I := 1 to Arraylen do
   23     3       begin  N[I] := I;  write(I: 2);  end;
   24     5     writeln;  write('Left,Right? ');
   25     7     readln(Left, Right);
   26     8     I := 4;
   27     9     Rotate(Left, Right, N);
   28    10     for I := 1 to Arraylen do
   29    11       write(N[I]: 2);
   30    12     writeln
   31         end.

*** No lines with errors detected ***
```

ROTAT.PAS is worth studying for a moment because it appears frequently throughout the remainder of this guide. The program prints an array of seven integers, then prompts you, asking for a starting and ending point in the array. Once the two input numbers have been entered, the program is supposed to rotate that section of integers to the left, with the left digit replacing the right. In its current form, the program compiles without problem, but as is shown in several of the following sections, its execution encounters numerous run-time errors.

Controlling the Debugger

Debugger takes control of the program, enters the command mode, and prompts with a right brace '}' symbol. (This may print on upper-case-only terminals as the right bracket ']' character.)

```
.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

}
```

You control the Debugger through single-character commands that generally take one of two forms, depending on whether or not the command accepts parameters:

} *single-character command*
} *single-character command* (*parameter(s)*)

Debugger commands and their parameters may be typed in either upper or lower case.

Command Syntax

In general, Debugger commands are used for controlling breakpoints, program execution, program tracking, data, and for displaying information about the data being maintained by the Debugger. Debugger commands can be stored in series and executed at designated locations within a program. Such locations are known as breakpoints and are specified by the break command. At any breakpoint, you may enter as many stored commands as fit on a single line. Any Debugger command may appear in a stored command and certain utility commands, described later, allow macros to be defined that let you store combined commands.

As an example of the general use of Debugger commands and the syntax for writing stored commands, look at the following command line. The line begins with the single-character command B followed by the parameter ROTATE,3. These direct the Debugger to set a breakpoint at statement 3 in procedure ROTATE. Next, a stored command is used to instruct the Debugger to write (W) the values of the variables I and A[I]. Stored commands are specified by placing them within angle brackets (<...>) after a break command and separating them by semicolons.

```
} B(ROTATE,3) <W(I); W(A[I])>
```

The Debugger accepts any of the single-character commands defined in the following sections. Numeric parameters in these sections are indicated by '*n*', as in the command S(*n*). A summary of the Debugger commands is given in Appendix A at the end of this guide, and the ? (question mark) command prints a similar list on your terminal screen.

Exiting and Stopping the Debugger

To exit from the Debugger at the prompt, give the command **Q** (quit), or type a Control-Z (^Z), or type Control-C (^C) twice in a row. A single Control-C (^C) typed during program execution stops the Debugger, thus permitting you to break into "infinite loops" in your program.

Selective Debugging

For certain large programs, you may wish to selectively debug portions of a program in order to speed up the debugging process or to reduce the amount of memory overhead created by the Debugger. You can edit your program to turn off generation of debugging information around procedures that have already been tested and debugged by using the embedded directives **$nodebug** and **$debug**. To turn off debugging, place the directive **$nodebug** before the procedure definition and the directive **$debug** after the procedure. **$Nodebug** and **$debug** are effective only when the **/debug** switch is first specified in the compilation command line. Otherwise they are ignored by the compiler. (See the User Guide for further details on embedded directives.)

Breakpoint Commands

Breakpoint commands allow you to set or remove breakpoints when your program reaches a certain point in execution or when a specified variable in your program changes value. Breakpoints allow you to interrupt the program in order to execute other Debugger commands.

B, B(): Control Breakpoints

A program control breakpoint is identified by two items: a block name (procedure, function, or **MAIN**), and a statement number within that block. For example, **ROTATE,3** identifies the third statement in the procedure **ROTATE**. Statements are sequentially numbered within each block. Statement numbers are listed in the second column of the program listing produced by the **debug** switch.

The **B**(*block,stmtnum*) command sets a control breakpoint within the block named *block* at the statement numbered *stmtnum*. When the breakpoint is reached, your program is interrupted before execution of the named statement, the breakpoint is identified, and the Pascal source line is displayed. The Debugger then accepts commands.

These may be interactive commands (from your terminal) or stored commands executed automatically. Any Debugger command may be stored for execution at a breakpoint. Stored commands are executed before interactive commands. If the stored commands direct the Debugger to resume execution, the program continues without waiting for an interactive command.

You may interrupt the program at any time with a Control-C (^C). This command stops the program and identifies the point of interruption as if you had set a control breakpoint.

NOTE

> If you type a Control-C (^C) while the program is awaiting input for a **real** or an **integer** at a **read** or **readln** statement, the Control-C (^C) does not take effect until after you have completed the input request.

A run-time error or program termination also causes a control breakpoint after the error message or termination status is displayed. You may set any number of control breakpoints. (The program executes more slowly if you define many.)

To set breakpoints in external functions and procedures, see "Debugging External Modules" later in this guide.

You may remove a breakpoint in two ways. The **B** command with no parameters deletes the breakpoint that most recently stopped the program. Otherwise, the **K** command described next may be used. (For uses of the **B** command, see the example listed after the **C** command.)

K, K(): Killing of Breakpoints

The **K**(*block,stmtnum*) command deletes the breakpoint specified by its parameter; the **K** command with no parameters removes all breakpoints. (See the example after the **C** command.) To remove breakpoints in external functions and procedures, see "Debugging External Modules" later in this guide.

V, V(): Data Breakpoints (Variables)

The data breakpoint facility (also called the "watched variable" command) causes an immediate breakpoint when the value of a specified variable is changed. The **V**(*variable*) command sets a data breakpoint, with *variable* indicating the variable to be monitored. When the value of the variable is changed, the Debugger prints both the old and new values and interrupts program execution for commands. Like control breakpoints, data breakpoints may have stored commands that are automatically executed when the breakpoint is triggered. A list of the stored commands, separated by semicolons, is enclosed in angle brackets after the watched variable command: **V**(*variable*)< ... >.

The **V** command monitors a variable of any type, but only the first 32 bytes of data is watched. You may watch any number of variables. (The program executes slowly if you set many.) For variables defined locally to a procedure, the watch command can either be set from within the procedure or through use of the **E** command defined later in this guide.

```
} B(ROTATE,1)<V(A[6])>  ———————— variable watch set within ROTATE
} G  ———————————————————————————— begin execution
Left,Right? 2 6  ———————————————— input to ROTAT
Breakpoint at ROTATE,1    begin
} C  ———————————————————————————— continue execution
The value of 'A[6]' was changed by the statement:
ROTATE,3  A[I] := A[I + 1];
Old value: 6
New value: 7
Breakpoint at ROTATE,4    A[Last] := A[First];
}
```

If a local variable is being monitored and the associated block is completed, the Debugger removes the breakpoint and displays a message that the variable no longer exists.

```
Breakpoint at ROTATE,1  begin  ———————— previously set breakpoint
} L  ———————————————————————— lists statements of procedure ROTATE
   14    1     begin
   15    2       for I := First to Last do
   16    3         A[I] := A[I + 1];
   17    4       A[Last] := A[First];
   18    5       write('Rotated ', first: 1, ' thru ', last: 1, '=');
   19          end;
} V(A[2])  ———————————————————— variable watched within ROTATE
} G
The value of ''A[2]'' was changed by the statement:
ROTATE,3  A[I] := A[I + 1];
Old value: 2
New value: 3
Breakpoint at ROTATE,3  A[I] := A[I + 1];
} G
Watch terminated for ''A[2]'' Value did not change.
Rotated 2 thru 6= 1 3 4 5 6 3 7  ———————— indicates a run-time error
}
```

This example gives us our first indication of a problem in the program ROTAT.PAS. The correct rotation for the starting and ending points (2,6) should read 1 3 4 5 6 2 7 not 1 3 4 5 6 3 7. Correction of the problem is explained in the section "Stepping Through a Debugging Session" later in this guide.

The **V** command without parameters removes all data breakpoints. It is not possible to remove individual data breakpoints.

Execution Control Commands

Execution control commands provide the means to monitor and control the flow of the program. The commands initiate, interrupt, or continue execution.

G: Go

The G (Go) command begins executing the program at MAIN,1 and may be used at any point in the program to restart it.

See the example after the C command.

C, C(): Continue Execution

The C (Continue) command resumes program execution from the current breakpoint.

If you set a breakpoint inside a loop, you may use the C(*n*) command to let the statement at the breakpoint execute *n* times. For instance, you may set a breakpoint at COUNT,10 inside a loop structure. When the Debugger stops at that breakpoint, you may give the command C(6) to let the loop iterate six times before the program stops again at COUNT,10. Each breakpoint has its own counter, which is independent of the counters for other breakpoints.

The C command functions like the G command to begin executing the program if you are at the start of the program.

If you use the C command after the program has terminated, you receive an error message telling you to use the G command to restart the program.

Examples of the B, K, D, G and C Commands

```
.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L(main,8,2)  ———————————————— List 8th statement of MAIN, 2 lines
   26    8      I := 4;
   27    9      Rotate(Left, Right, N);
} B(main,9)<W('I=',i);C>
} B(Rotate,3)<W('In rotate, I=',i)>
} G
 1 2 3 4 5 6 7
Left,Right? 2 6  ———————————————————————— input to ROTAT
Breakpoint at MAIN,9  Rotate(Left, Right, N);
I=  4
Breakpoint at ROTATE,3  A[I] := A[I + 1];
In rotate, I=  2
} D  ———————————————————————————————— display breakpoints

Breakpoints

ROTATE,3  A[I] := A[I + 1];
      <W('In rotate, I=',I)>

MAIN,9  Rotate(Left, Right, N);
      <W('I=',I);C>
```

```
} W(I); C(2); W(I)
2
Breakpoint at ROTATE,3  A[I] := A[I + 1];
In rotate, I=  4
4
} K(Rotate,3)  ———————————————————— kill specified breakpoint
} C  ———————————————————————————— continue execution
Rotated 2 thru 6= 1 3 4 5 6 3 7  ———————— indicates run-time error
} D  ———————————————————————————— display breakpoints

Breakpoints

MAIN,9  Rotate(Left, Right, N);
     <W('I=',I);C>

} K  ———————————————————————————— kill all breakpoints
} D  ———————————————————————————— (no breakpoints to display)
} Q  ———————————————————————————— quit the Debugger
```

S, S(): Step to Next Statement

The S (Step) command executes the next statement of the program. The S(*n*) command executes *n* statements without interruption. If a statement being "stepped" calls another procedure or function, that new procedure or function also is executed one step at a time.

See the example after the P command.

P, P(): Proceed to Next Statement

The P (Proceed) command executes the next statement at the current level of the program. P differs from S in that P does not single-step through functions and procedures called by the current statement. P treats an entire nested call as a single statement; thus procedure calls and function invocations are completed before program control returns to the Debugger, allowing you to bypass the detailed execution of routines (e.g., ones already debugged). If the current procedure ends, P begins single-stepping the procedure that called the current procedure.

The P(*n*) command is equivalent to repeating the P command *n* times.

As with the C command, you may not go past the end of the program with an S or a P command. If you do so, you receive an error message telling you to use G to restart the program.

Examples of the S and P Commands

```
.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983
Debugging program ROTAT

} B(main,9)
} G
 1 2 3 4 5 6 7
Left,Right? 1 5  ———————————————————— input to ROTAT
Breakpoint at MAIN,9  Rotate(Left, Right, N);
} S
Breakpoint at ROTATE,1  begin;
} S
Breakpoint at ROTATE,2  for I := First to Last do
} S
Breakpoint at ROTATE,3  A[I] := A[I + 1];
} S
Breakpoint at ROTATE,3  A[I] := A[I + 1];
} S(4)
Breakpoint at ROTATE,4  A[Last] := A[First];
} G
Rotated 1 thru 5= 2 3 4 5 2 6 7  ——— further indication of run-time error
} G
 1 2 3 4 5 6 7
Left,Right? 1 5  ———————————————————— input to ROTAT
Breakpoint at MAIN,9  Rotate(Left, Right, N);
} P
Breakpoint at MAIN,10 for I := 1 to Arraylen do
} P
Breakpoint at MAIN,11  write(N[I]:2);
} P
Breakpoint at MAIN,11  write(N[I]:2);
} P
Breakpoint at MAIN,11  write(N[I]:2);
} P(6)
Rotated 1 thru 5= 2 3 4 5 2 6 7  ————— same indication of a problem
}
```

Tracking Commands

Two commands help you track program execution. The H command lists the statements that have brought you to your present position. The T command traces program execution through each statement.

H, H(): History of Program Execution

The Debugger maintains a list of the last 50 statements executed while your program was running. With the H command you may review this execution history. For instance, if the program failed because of an error during execution (such as division by zero), the H command shows the steps leading to the statement causing the error. The H command with no parameters prints a list of the last 10 statements executed. H(*n*) prints the last *n* statements up to 50.

The H command has other important functions as well. See "Execution Stack Commands" for details and for examples of the command.

T(): Execution Trace

The T command accepts a Boolean parameter, either enabling or disabling the tracing of program execution. When tracing is enabled with the T(TRUE) command, each statement is identified by its block name and statement number and is displayed before being executed.

A Control-C (^C) interrupts the trace and returns the Debugger to command mode. You may then turn off tracing with the T(FALSE) command and continue running your program with the C command.

Example of the T Command

```
.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L(main,9,3)
   27    9      Rotate(Left, Right, N);
   28   10      for I := 1 to Arraylen do
   29   11        write(N[I]:2);
} B(main,9)  ------------------------------------- set breakpoint
} G
 1 2 3 4 5 6 7
Left,Right? 1 3  ------------------------------- input to ROTAT
Breakpoint at MAIN,9  Rotate(Left, Right, N);
} T(TRUE)
} G
```

```
ROTATE,1  begin ———————————————————— tracing output begins
ROTATE,2  for I := First to Last do
ROTATE,3  A[I] := A[I + 1];
ROTATE,3  A[I] := A[I + 1];
ROTATE,3  A[I] := A[I + 1];
ROTATE,4  A[Last] := A[First];
ROTATE,5  write('Rotated ',first: 1,' thru ',last: 1,'=');
MAIN,10   for I := 1 to Arraylen do
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,11   write(N[I]:2);
MAIN,12   writeln
Rotated 1 thru 3= 2 3 2 4 5 6 7  ——— our run-time error is still evident
} T(FALSE) ———————————————————— tracing off
} K
} G
 1 2 3 4 5 6 7
Left,Right? 1 3 ———————————————— input to ROTAT
Rotated 1 thru 3= 2 3 2 4 5 6 7
}
```

Data Commands

Debugger data commands allow you to display the current values of variables and to assign new values to them. The data commands provide full access to user identifiers and type definitions. The data commands conform to Pascal type compatibility rules.

▼(): Write Variable Value

You use the ▼ command to write the value of a variable (including a pointer), of a constant, or of a memory location. The format for the ▼ command is:

```
} ▼(name1,name2,name3, ...)
```

where *name* is the name of the variable you want written. As shown, you may write the value of more than one variable by separating variable names with commas.

The type of variable determines the format of the output. For example, integers are displayed as signed decimal numbers. Set variables are displayed in Pascal set notation. Scalar variables are displayed as the names of the enumerated types they represent.

You may use the Pascal colon notation ':' to alter the way variables are written. For example, to print the integer variable I as a hexadecimal number, you use:

```
} ▼(I:-1)
```

Also see the example after Variable Assignment.

Real numbers may be formatted according to the same rules used by the compiler.

A numeric constant is used as an address if you wish to write the integer value contained in a memory location. A 'B' placed after the number, as in ▼(27740B), specifies an octal memory location. Memory locations are displayed as signed integers.

The Debugger may write any complex Pascal data structure, including records and arrays, except files.

The Debugger displays an array in an orderly fashion that reflects the array's structure. For each change in the least significant (rightmost) index of the array, the Debugger writes a space between elements. For each change in the next least significant (second-from-rightmost) index, the Debugger starts a new line. And for changes in the *nth* index, where n is the number of "places from the right" of the least significant index and n is greater than 2, the Debugger writes $n - 2$ blank lines and indents the first row of the display $n - 2$ spaces.

Example:

```
Pascal-2 RT11 SJ V2.1D   9-Feb-84   7:06 AM     Site #1-1     Page 1-1
Oregon Software, 6915 SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202
MULTI/DEBUG

 Line Stmt
    1      program Multi;                    { multidimensional variables }
    2
    3      var A: array [1..3, 1..3, 1..3] of integer;
    4          I, J, K: integer;
    5
    6    1 begin
    7    2   for I := 1 to 3 do
    8    3     for J := 1 to 3 do
    9    4       for K := 1 to 3 do
   10    5         A[I,J,K] := (I * 10 + J) * 10 + K;
   11      end.

 *** No lines with errors detected ***


.RUN MULTI

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program MULTI

} G
} W(A)
 111 112 113
121 122 123
131 132 133

 211 212 213
221 222 223
231 232 233

 311 312 313
321 322 323
331 332 333
```

When you write records, the Debugger lists each field name followed by the value of that field. The format of each field is determined by the data type of the field. Complex records, such as those containing arrays of records, can get messy; you may want to have the listing on hand to show the definition of the record being printed.

Variable Assignment

The Debugger command to modify the value of a program variable is identical in form to a Pascal assignment statement. The left-hand side of the ':=' assignment operator indicates the variable to be modified. This variable may include array indices, record field selectors, and pointer accesses. The right-hand side specifies the value to be assigned. This may be a simple constant or literal value, or another program variable. Standard notation is used for all values, including sets. General expressions (operators and functions) are not permitted.

Debugger variable assignments must conform to the Pascal assignment compatibility rules. All variables accessed in an assignment command must be available in the current stack context. The E(*n*) command may be used to temporarily change context, if necessary.

Examples of the W Command and Variable Assignment

```
Pascal-2 RT11 SJ V2.1D   9-Feb-84   7:06 AM      Site $1-1      Page 1-1
Oregon Software, 6915 SW Macadam Ave., Portland, Oregon 97219, (503) 245-2202
COLOR/DEBUG
 Line Stmt
    1       program Color;
    2
    3         type
    4           Color = (Red, Orange, Yellow, Blue, Green);
    5
    6         var
    7           c: Color; I: integer;
    8           Colorset: set of Color;
    9           a: array [0..4] of Color;
   10           r: record
   11                I: integer;
   12                S: set of Color;
   13                C: packed array [1..4] of char;
   14              end;
   15
   16     1    begin
   17     2      for C := Red to Green do A[ord(C)] := C;
   18     4      Colorset := [Red, Yellow..Green];
   19     5      R.I := 123; R.S := [Orange, Green]; R.C := 'TEST';
   20          end.

 *** No lines with errors detected ***

.RUN COLOR

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program COLOR

} G
} W(A)
 RED ORANGE YELLOW BLUE GREEN

} A[1] := Red; A[4] := Red; W(A)
 RED RED YELLOW BLUE RED
```

```
} W(Colorset)
[RED,YELLOW..GREEN]
} Colorset := [Red..Green]; W(Colorset)
[RED..GREEN]
} W(R)
I: 123
S: [ORANGE,GREEN]
C: TEST

} R.I := 321; R.S := Colorset; W(R)
I: 321
S: [RED..GREEN]
C: TEST

}
```

Informational Commands

Informational commands show data being maintained by the Debugger. The D command shows the current breakpoints, user-defined macros, and variables being watched. The L command shows selected parts of the program listing, so that you won't have to reprint the listing each time you revise your program.

D: Display Parameters

The D command displays all breakpoints, user-defined macros, and the variables being watched; it also shows any commands associated with each. Breakpoints are set with the B command. Macros are stored Debugger commands created by the M command and executed by the X command. The W command is used to set variable watches. (See the respective sections for details on these commands.)

See the ROTAT.PAS example in "Running the Debugger" and the example after the C command.

L, L(): List Source Lines

The L command uses the statement numbers in the listing file of your program to list portions of the source program. The L command allows you to list individual statements, parts of procedures, or entire procedures.

When a breakpoint is set at a statement with B(), the Debugger prints only the first line associated with the statement. The History command H also prints only the first line of the statement. The L command, in contrast, prints all lines containing the statement.

The L command with no parameters lists the current procedure. You may list any other procedure by giving the procedure name enclosed in parentheses. For example, L(MAIN) lists the body of the main program.

The command L(*proc*,*stmtnum*) lists a single statement, where *proc* is the name of the procedure and *stmtnum* is the number of the statement to print.

You also may list sections of the program starting or ending at a particular statement by specifying a line count after the statement number. For example, L(MAIN,1,10) lists the first ten lines of the main program.

The general form of the command is:

```
} L(proc,stmtnum,count)
```

where *proc* and *stmtnum* describe a statement in the program. A positive *count* prints that many lines starting at the statement specified and moving forward. A negative *count* causes the Debugger to list statements up to and including *stmtnum*. (The listing of source lines in external functions and procedures requires a slightly different form of the L command. See "Debugging External Modules" later in this guide for details.)

This example lists 5 lines beginning with the first statement of procedure ROTATE:

```
} L(Rotate,1,5)
  14    1       begin
  15    2         for I := First to Last do
  16    3           A[I] := A[I + 1];
  17    4         A[Last] := A[First];
  18    5         write('Rotated ',first: 1,' thru ',last: 1,'=');
```

This example lists 2 lines leading up to and including the 4th statement of procedure Rotate:

```
} L(Rotate,4,-2)
  16    3          A[I] := A[I + 1];
  17    4        A[Last] := A[First];
```

When you list an entire procedure, the Debugger attempts to include the procedure heading and local variable declarations in the listing. However, this header information is only used by the Pascal compiler, so the Debugger has to estimate where the procedure header information is located in the listing file. As a result, the Debugger may not always print the complete header information or may sometimes print part of the preceding procedure.

Long procedures may take some time to print. A single Control-C (^C) interrupts the listing and returns the Debugger to command mode.

Utility Commands

Utility commands allow you to define a series of commands as a macro to be executed by entering a single command.

M(): Define Macro

The M command saves you some typing when you need to issue repetitive commands. For example, you may need to write the value of several critical variables at different places in your program. The M feature lets you combine these commands under one name, then execute this group of commands by using the X command, explained below. You cannot pass parameters to macros.

The format for definition of a macro is:

```
} M(name)<command1; command2; command3; ...>
```

where *name* is any alphanumeric string containing up to 32 symbols. The X command uses *name* to identify the macro. You may place as many Debugger commands in the angle brackets '< >' as fit on one command line. You may delete a macro by typing M(*name*) with no commands. Available memory is the only limit on the number of macros you may define. The D command lists macro names and the commands associated with each name.

See the example after the X command.

X(): Execute Macro

You may execute the Debugger commands associated with a macro by using the X command. The format is:

```
} X(name)
```

where *name* is the name of the macro. The effect of the X command is to execute the Debugger commands defined by the M command of that name.

Examples of the M and X Commands

```
.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} M(DumpN)<W('The value of N=',N)>  ———————————— define macro
} B(Rotate,1); G
 1 2 3 4 5 6 7
Left,Right? 2 6  ———————————————————————— input to ROTAT
Breakpoint at ROTATE,1  begin
} M(DumpI)<W('I=',I)>  ———————————————————— define macro
} D
Breakpoints

ROTATE,1  begin

Macros

DUMPI       W('I=',I)
DUMPN       W('The value of N=',N)
} S
Breakpoint at ROTATE,2  for I := First to Last do
} S
Breakpoint at ROTATE,3  A[I] := A[I + 1];
} X(DumpI)  ———————————————————————————— execute macro
I=  2
} S(4); X(DumpI); X(DumpN)
Breakpoint at ROTATE,3  A[I] := A[I + 1];
I=  6
The value of N=   1 3 4 5 6 3 7

} Q
```

Execution Stack Commands

The execution stack commands allow you to trace down run-time errors by examining the stack. The H command shows at any time a history of program execution and the current stack of active procedure and function calls. The N command lists the names of the parameters and local variables in any procedure in the execution stack. The E command allows you to change the context of the stack frame from the current procedure to another so you may access variables you otherwise wouldn't be able to.

H, H(): History of Program Execution

The Debugger maintains a list of the last 50 statements executed while your program was running. With the H command you may review this execution history. For instance, if the program failed because of an error during execution (such as division by zero), the H command shows the steps leading to the statement causing the error. The H command with no parameters prints a list of the last 10 statements executed. H(*n*) prints the last *n* statements up to 50.

The H command also lists the execution stack. Each time a procedure or function is called, a new entry is made at the top of the execution stack. When the procedure exits, that entry is removed from the top of the stack. The main program is always at the bottom of the stack. The H command shows the procedures that were called to get from the main program to the current procedure. H(0) prints just the execution stack.

In the display, each procedure or function in the execution stack is identified by a number. This procedure number is used to identify procedures in the execution stack for the N and E commands described in following sections. (These are **not** the statement numbers used to identify other Debugger commands.)

In the display, the '<' character marks the current procedure. Unless the E command is used the current procedure is always the top procedure in the execution stack. The Debugger uses the current procedure to determine the local variables that can be accessed according to Pascal scope rules. Procedures marked with the asterisk '*' character are those procedures that contain the lexical definition of the current procedure. The parameters and local variables in the procedures marked by '<' or '*' are the only local variables that you may look at or change directly. If you wish to look at local variables in other procedures in the execution stack, you must use the E command.

See the example after the E command.

V, V(): Names of Variables

The V command with no parameters lists the names of the parameters and local variables in the current procedure. If you are in the main program, the command displays all of the global-level variable names.

V with a numeric parameter lists the names of the local variables in the procedure so numbered on the execution stack. These numbers are obtained via the H command, described above.

Note that V lists the names of the local variables and parameters in any procedure or function on the stack, not merely the ones marked with the '*'. However, you cannot write or change the values of variables unless they are in procedures or functions marked with the '*'.

The E command allows access to variables that you otherwise cannot access from the current procedure.

See the example after the E command.

E(): Enter Stack-Frame Context

The Debugger normally enforces Pascal scope rules. If you stop your program in the middle of a procedure, you may write or modify only those variables and parameters of the procedures that enclose the current procedure, as described in the section on the H command.

To look at or change local variables in procedures that are not accessible to the current procedure, the E command gets around the Pascal scope rules by temporarily changing the context of the current procedure.

The H command numbers the procedures in the execution stack. The main program is always 1, and procedures called from the main program are listed as 2, and so on. If you want to examine variables in procedure 5 in the current execution stack, and it is not marked with an '*' (and therefore not available to you from where you are), you use E(5) to temporarily enter the context of that procedure.

The E command affects only debugging commands that follow it on the same command line. For example, to print the value of the variable I in the procedure listed as 5, you type:

```
} E(5): V(I)
```

This command line makes procedure 5 the current procedure. Then, using the context of procedure 5, the Debugger prints the value of the variable I. At the end of the command line, the current procedure is changed back to the top procedure in the execution stack.

Because the V command allows you to list the names of variables in all the procedures on the execution stack, the following commands are equivalent:

```
} E(5): V
} V(5)
```

Examples of the H, N, and E Commands

```
Breakpoint at CHECK,1  begin   { start of check }
} H(5)  ———————————————————————— list last 5 statements executed

Program execution history:

ANALYZEMOVE,9  Vacant[Target] := false;
ANALYZEMOVE,10  if CentralSquares[Target] then
ANALYZEMOVE,14  PossibleMoves := PossibleMoves+1;
ANALYZEMOVE,15  Check(4); Check(5); Check(-4); Check(-5);
CHECK,1  begin   { start of check }

Procedure execution stack

 8< CHECK,1  begin   { start of check }
 7* ANALYZEMOVE,15  Check(4); Check(5); Check(-4); Check(-5);
 6* ANALYZE,12  AnalyzeMove(4,I); AnalyzeMove(5,I);
 5* EVALUATEBOARD,4  Analyze;
 4  GENMOVE,15  EvaluateBoard(N^,Turn);
 3  MOVEPIECE,9  if MovesAllowed then GenMove(I,J);
 2  EXPAND,11  if Color[Who]=Turn then MovePiece(I,I,0,0);
 1* MAIN,8  Expand(Root,True);

} N  ———————————————————————————————— local names
DIRECTION SRC DST F
} N(7)  ———————————————————————————— names in frame 7
DIRECTION I SAFE MASKING TARGET THRT
} N(4)  ———————————————————————————— names in frame 4
I J N OLDPIECE
} E(7): W(I)  ———————————— change context to frame 7, write value
14
} E(4): W(I)  ———————————— change context to frame 4, write value
27
} E(4): H(0)

Procedure execution stack

 8  CHECK,1  begin   { start of check }
 7  ANALYZEMOVE,15  Check(4); Check(5); Check(-4); Check(-5);
 6  ANALYZE,12  AnalyzeMove(4,I); AnalyzeMove(5,I);
 5  EVALUATEBOARD,4  Analyze;
 4< GENMOVE,15  EvaluateBoard(N^,Turn);
 3* MOVEPIECE,9  if MovesAllowed then GenMove(I,J);
 2* EXPAND,11  if Color[Who]=Turn then MovePiece(I,I,0,0);
 1* MAIN,8  Expand(Root,True);

}
```

Stepping Through a Debugger Session

You seldom use only a single Debugger command at any one session, so no single-command example can demonstrate the context in which certain commands are used nor can it demonstrate all of the ways in which certain commands relate. Our approach, therefore, is to step through a sample program to demonstrate some of the common commands in a problem/example context.

In previous sections of this guide, several examples of run-time errors occurring in the execution of ROTAT.PAS were demonstrated. Let's begin this debugging session by and running the program and taking a closer look at what is going wrong.

```
.R PASCAL
*ROTAT/DEBUG

.LINK ROTAT.SY:PASCAL

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

}
```

After the Debugger has taken control of the program, we instruct it to go (G), then we enter the starting point (2,6).

```
} G
 1 2 3 4 5 6 7
Left,Right? 2 6
Rotated 2 thru 6= 1 3 4 5 6 3 7
```

The problem we noted earlier persists. Our starting number in the rotation is apparently incremented by 1 each time the program is run. In this case, the second 3 in our rotated sequence should be 2. With the L command, we now list the part of the main program that initializes the N array. From this, we can choose a location for a breakpoint once the array is initialized.

```
} L(main.1.5)  ———————————————————— list 5 lines of main program
   21    1    begin   { Main program }
   22    2      for I := 1 to Arraylen do
   23    3        begin N[I] := I; write(I:2); end;
   24    5      writeln; write('Left,Right? ');
   25    7      readln(Left, Right);
} B(main.6)  ———————————————————— set breakpoint at MAIN,6
} G  ———————————————————————————— begin execution
 1 2 3 4 5 6 7
Breakpoint at MAIN,6  writeln; write('Left,Right? ');
} W(N[6])  ———————————————————— write value of N[6]
6
```

(Note the way in which the Debugger counts statements when more than one is placed on a line, as on line number 24 above. Though not explicitly listed, the second statement on line 24 is statement number 6 and must be identified as such.)

Examination of the array N at this breakpoint shows the array to be correct; the change to the value of the variable must be occurring somewhere else. Using the W (watched variable) command, we tell

the Debugger to stop the program whenever A[6] is changed.

```
} W(A[6])  ———————————————— watch for changes of value of A[6]
} C  ———————————————————— continue execution
Left,Right? 2 6  ———————————— input to ROTAT
The value of 'A[6]' was changed by the statement:
ROTATE,3  A[I] := A[I + 1];
Old value: 6
New value: 7
Breakpoint at ROTATE,4 A[Last] := A[First];
}
```

This is an expected change based on the algorithm being used in the rotation. Our last number is first incremented by 1 before being replaced by the first. Therefore, we continue to watch the variable.

```
} C
The value of 'A[6]' was changed by the statement:
ROTATE,4  A[Last] := A[First];
Old value: 7
New value: 3
Breakpoint at ROTATE,5 write('Rotated ',first:1,' thru ',last:1,'=');
} W(First,Last)  ———————— write values of First and Last
2  6
} W(A)  ———————————————— write values of array A
 1 3 4 5 6 3 7

} Q  ———————————————————— quit the Debugger
```

At ROTATE,4 the first element is assigned to the last element after the first element has already been changed. We must introduce a temporary variable to hold the first element value so that it will not be destroyed. We correct the program (adding a "temp" variable, an assignment at line 14 and another between lines 16 and 17), then recompile with the /debug switch. Again, we use the L command to inspect the part of the program we changed.

```
Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L(ROTATE,1,6)  ———————— list 6 lines of procedure ROTATE
   14    1     begin Temp := A[First];
   15    3       for I := First to Last do
   16    4         A[I] := A[I + 1];
   17    5       A[Last] := Temp;
   18    6       write('Rotated ',first: 1,' thru ',last: 1,'=');
   19          end;
} G  ———————————————————— begin execution
 1 2 3 4 5 6 7
Left,Right? 2 6  ———————————— input to ROTAT
Rotated 2 thru 6= 1 3 4 5 6 2 7
} G  ———————————————————— begin execution
 1 2 3 4 5 6 7
Left,Right? 3 4  ———————————— input to ROTAT
Rotated 3 thru 4= 1 2 4 3 5 6 7
Breakpoint at MAIN,12  writeln
```

Now the program seems to be running correctly, but let's make one more test before we've satisfied ourselves that the program is running as we want. Note that the G command restarts the program even after it has terminated.

```
} G                                                    ——— begin execution
 1 2 3 4 5 6 7
Left,Right? 1 7                                        ——— input to ROTAT

%T2 -- Fatal error at user PC=1424
Array subscript out of bounds

}
```

The end points of a data subrange are always good places to look for run-time errors such as "Array subscript out of bounds," because they are the values most likely to exceed the predefined limits. We begin analyzing this new error by writing the values of variables found in the line where the error occurred.

```
} W(I)                                                 ——— write the value of I
7
} W(A[8])                                              ——— write the value of A[8]
%Array subscript too large
W(A[8])
    ^
%The limits are 1..7
} Q                                                    ——— quit the Debugger
```

Now we can diagnose the error. We see that the limits for the array subscript of A have been exceeded by one. The **for** loop in the **ROTATE** procedure is likely to be looping too many times. We reduce the final value by 1 (**last** becomes **last-1** in line 15) and recompile the program. When we run the program, we tell the Debugger to list procedure **ROTATE**, so that we can more closely follow the section of the program we changed.

```
Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L(ROTATE,1,6)                                        ——— list 6 lines of procedure ROTATE
   14     1      begin Temp :=A[First];
   15     3        for I := First to Last - 1 do
   16     4          A[I] := A[I + 1];
   17     5        A[Last] := Temp;
   18     6        write('Rotated ',first: 1,' thru ',last: 1,'=');
} G                                                    ——— begin execution
 1 2 3 4 5 6 7
Left,Right? 1 7                                        ——— input to ROTAT
Rotated 1 thru 7= 2 3 4 5 6 7 1
} Q                                                    ——— quit the Debugger
```

The results our program gives are now exactly what they should be. Once satisfied that the program is correct, we recompile it without the **debug** switch to reduce file and memory requirements, to improve execution speed, and to reinstate the walkback.

Debugging External Modules

An external module consists of one or more Pascal procedures or functions written and compiled independently of the main program that invokes it. The Debugger's ability to debug external modules allows you to fully debug an entire program, including externals, and also allows you to debug external procedures and functions only, in the context of a main program.

The debugging of external procedures and functions is simply a matter of compiling the module with the **debug** and **nomain** switches, linking the module with the main program, and upon execution, supplying the module name on certain Debugger commands (see below). The **debug** compilation creates the necessary symbol table files for the module. (Remember to compile the main program with the **debug** switch, too.) Refer to "External Modules" in the Programmer's Guide for rules on the use of external modules.

As mentioned earlier, external modules cannot be debugged directly; they must be called from a main program. To debug an external procedure or function itself, create a short main program that simply invokes the procedure, then exits. Be sure to initialize any variables required of the call. Then compile the main program using the **debug** switch and task build it with the external module.

Differences in the Commands

This section covers only the differences in command syntax and usage; unless otherwise noted, the Debugger commands work as described in previous sections.

In general, the major differences are:

- The **B**, **K** and **L** commands accept the module name followed by a colon (:) as the first argument. These commands allow you to set and kill breakpoints and list source lines in external procedures and functions. The revised syntax for these commands are as follows:

  } B(*module*:*block*,*stmtnum*)< ... >
  } K(*module*:*block*,*stmtnum*)
  } L(*module*:*block*,*stmtnum*,*count*)

  The module name *module* is the name of the source file minus extension. For example, **TEST** is the module name for TEST.PAS. *Block* is the name of the procedure or function being referenced in *module*. The other arguments are the same as described in earlier sections.

- When displaying breakpoint and source-line information, the Debugger includes the module name along with the procedure name and line number. The **D** command, in addition to displaying breakpoints, user-defined macros, and variables being watched, shows you which module is currently being debugged.

- Defaults apply to the current module being debugged. To list lines, set breakpoints or kill breakpoints in any module but the current one, you must specify at least the module name and the procedure name on the commands.

- If you try to list lines or set/kill breakpoints in an external module **not** compiled for debugging, the run-time error "can't open file" causes the Debugger to abort trying to open the listing and symbol table files for that module. Of course, if you have old listing and symbol table files for that module lying around, the Debugger opens these files even if you did not wish to debug that module. In this case, if the data files do not match the source you're using, your results may not be accurate.

An example is presented in the next section.

Example

To illustrate the debugging of external modules, we present a single debugging session in which the above commands are used. In this example, the main program ROTAT.PAS, presented earlier, calls the external procedure **Rotate** contained in XROT.PAS. (In the previous program, procedure **Rotate** was a local procedure.)

After compiling the main program and external modules and task building them, run the program.

```
.R PASCAL
*ROTAT/DEBUG

.LINK ROTAT,ROTATE,SY:PASCAL
.RUN ROTAT

Pascal Debugger V3.00 -- 29-Nov-1983

Debugging program ROTAT

} L  ------------------------------------------ defaults to main program
   14     1   begin { main program }
   15     2     for I := 1 to Arraylen do
   16     3       begin  N[I] := I;  write(I: 2);  end;
   17     5     writeln;  write('Left,Right? ');
   18     7     readln(Left, Right);
   19     8     I := 4;
   20     9     Rotate(Left, Right, N);
   21    10     for I := 1 to Arraylen do
   22    11       write(N[I]: 2);
   23    12     writeln
   24         end.
} B(MAIN,7)
} B(XROT:ROTATE,5)<W(TEMP)>
```

Initially, the **L** command without parameters lists the main program by default because it is in the current module being debugged. The first breakpoint command could just as easily be **B(ROTAT:MAIN,7)**. However, the module name is not necessary because the main program is the current module. The second breakpoint command shows that you must supply the module name for modules other than the current one. With the **G** command, program execution begins:

```
} G
 1 2 3 4 5 6 7
Breakpoint at ROTAT:MAIN,7  readln(Left, Right);
} G
Left,Right? 1 7  ------------------------------- input to ROTAT
Breakpoint at XROT:ROTATE,5  A[Last] := Temp;
1
```

Note the way the Debugger reports the breakpoints. At this point the current procedure being debugged is the procedure **Rotate**, in external module XROT. (The single '1' is the value of **Temp** when the breakpoint is reached.) Now the **B**, **K** and **L** commands default to the external procedure **Rotate**, as shown below for the **L** command. To list the lines in the main program, you must specify the module name, as shown in the second **L** command:

```
} L
   16     1 begin  Temp := A[First];
   17     3   for I := First to Last-1 do
   18     4     A[I] := A[I + 1];
   19     5   A[Last] := Temp;
   20     6   write('Rotated ', first: 1, ' thru ', last: 1, '=');
   21       end;
} L(ROTAT:MAIN)
   14     1   begin { main program }
   15     2     for I := 1 to Arraylen do
   16     3       begin  N[I] := I;  write(I: 2);  end;
   17     5     writeln;  write('Left,Right? ');
   18     7     readln(Left, Right);
   19     8     I := 4;
   20     9     Rotate(Left, Right, N);
   21    10     for I := 1 to Arraylen do
   22    11       write(N[I]: 2);
   23    12     writeln
   24         end.
} D  ------------------------------ display current module and breakpoints
Current module: XROT

Breakpoints

XROT:ROTATE,5  A[Last] := Temp;
     <W(TEMP)>

ROTAT:MAIN,7  readln(Left, Right);
} H  ------------------------------------------------ History command

Program execution history:

XROT:ROTATE,1  begin  Temp := A[First];
XROT:ROTATE,2  begin  Temp := A[First];
XROT:ROTATE,3  for I := First to Last-1 do
XROT:ROTATE,4  A[I] := A[I + 1];
XROT:ROTATE,4  A[I] := A[I + 1];
XROT:ROTATE,4  A[I] := A[I + 1];
XROT:ROTATE,4  A[I] := A[I + 1];
XROT:ROTATE,4  A[I] := A[I + 1];
XROT:ROTATE,4  A[I] := A[I + 1];
XROT:ROTATE,5  A[Last] := Temp;

Procedure execution stack

 3< XROT:ROTATE,5  A[Last] := Temp;
 2* Module: XROT
 1* ROTAT:MAIN,9  Rotate(Left, Right, N);
```

Execution continues with the C command. The K and B commands used below demonstrate the rules governing the setting and removing of breakpoints. Note the erroneous breakpoint command and the corrected command.

```
} C  ———————————————————————————————— continue execution
Rotated 1 thru 7= 2 3 4 5 6 7 1

Program terminated.

Breakpoint at ROTAT:MAIN,12  writeln
} K(MAIN,7)  ———————————————————————————— kill breakpoint
} G
 1 2 3 4 5 6 7
Left,Right? 1 7  ———————————————————————— input to ROTAT
Breakpoint at XROT:ROTATE,5  A[Last] := Temp;
1
} B(MAIN,7)  ———————————————— won't work without the module name
No such statement in this procedure
B(MAIN,7)
        ^
} B(ROTAT:MAIN,7)  ———————————————————————— that's better
} C
Rotated 1 thru 7= 2 3 4 5 6 7 1
} Q  ———————————————————————————————— quit the Debugger
```

Overlays

The debugging of large programs is aided by the execution of two command files, EXTRAC.COM and either XMDBG.COM (for XM systems) or SJDBG.COM (for SJ systems), which are included in the distribution kit. These command files produce a much smaller load image than that of a non-overlaid Debugger, saving up to 6K words of memory. These command files can be executed with the indirect (at-sign) processor.

The command file EXTRAC.COM extracts from the Pascal support library the modules needed for overlaying the Debugger against a Pascal program. EXTRAC.COM need only be executed once, at installation, to ensure that the necessary Debugger and support library modules are available. The modules are placed on the system device, with the extension .DBG.

The command file XMDBG.COM, used in the XM environment, links your program and the Debugger into virtual overlays. The command file SJDBG.COM, for the SJ environment, links your program and the Debugger into overlays.

After the modules have been extracted, examine and modify the link command file you wish to use, replacing all occurrences of FOO in the file with the name of your program. Then execute the link command file to create the overlaid program. The program is now ready to run.

When debugging programs that use overlays don't confuse the program's existing overlay structure with the structure imposed by the link command file you are using. The results may be unpredictable.

Appendix A: Debugger Command Summary

B	Remove current breakpoint
B(*block,stmtnum***)**	Set a control breakpoint
B(*block,stmtnum***)< ... >**	Control breakpoint with stored commands
PDP B(*module:block,stmtnum***)**	Set a control breakpoint in external module
B(*module:block,stmtnum***)< ... >**	Set external control breakpoint with stored commands
C	Continue program execution
C(*n***)**	Continue *n* times
D	Display breakpoints and macros
E(*n***)**	Enter context of frame *n* (1 line only)
G	Restart program
H	Display recent history and full stack
H(*n***)**	Display last *n* statements executed
K	Remove all control breakpoints
K(*block,stmtnum***)**	Remove specified breakpoint
K(*module:block,stmtnum***)**	Remove specified breakpoint from external module
L(*proc***)**	List source of *proc*
L(*proc,stmtnum***)**	List statement *stmtnum* in *proc*
L(*proc,stmtnum,x***)**	List *x* lines beginning with statement *stmtnum* in *proc*
L(*module:proc***)**	List source of external module *proc*
L(*module:proc,stmtnum***)**	List statement *stmtnum* in external module *proc*
L(*module:proc,stmtnum,x***)**	List *x* lines beginning with statement *stmtnum* in external *proc*
M(*name***)<***commands***>**	Define stored command macro
N	List variable names for current frame
N(*n***)**	List variable names for frame *n*
P	Proceed 1 statement at current level
P(*n***)**	Proceed *n* statements
Q	Quit Debugger
S	Single-step statement
S(*n***)**	Single-step *n* statements
T(TRUE/FALSE)	Enable/disable tracing
V(*variable***)**	Set data breakpoint
V(*variable***)< ... >**	Data breakpoint with stored commands
W()	Write list of values
X(*name***)**	Execute named macro command
variable* := *value	Assign *value* to *variable*
?	Help (display command summary)
^C (Control-C)	Immediate breakpoint
^Z (Control-Z)	Exit from Debugger

The Pascal–2 Profiler

The Pascal–2 Profiler can help you tune Pascal programs by detecting bottlenecks: small sections of code in which your program spends a disproportionately large amount of time. The Profiler counts the number of times each Pascal statement in your program is executed then prints a summary describing how many times each procedure is called and what percentage of the total statements executed are found in that procedure.

To use the Profiler, you should compile your program with the **profile** switch. (See the Programmer's Guide for details on compilation switches.) The **profile** switch causes the Pascal–2 compiler to generate several auxiliary files. These files, which permit the Profiler to locate the statements and procedures in your program, are the same ones generated by the **debug** switch.

The compilation and steps are shown below, using a program, CHECKR.PAS, which plays a game of checkers.

```
.R PASCAL
*CHECKR/PROFILE

.LINK CHECKR,SY:PASCAL
```

Upon execution, the Profiler takes control of the program and opens the program's auxiliary files created by the Pascal compiler. For large programs, there may be a short pause while the Profiler scans the auxiliary files to build internal data structures.

Next, the Profiler prompts for the name of the profile output file. If you specify a disk file, the default file extension is .PRO. Writing a profile to the terminal is practical only for a short program.

Compile and link the program as previously shown, then run it.

```
.RUN CHECKR

profile  V2.1B -- 6-Feb-1983

Profiling module: CHECKR

Profile output file name? CHECKR  ———— Output goes to CHECKR.PRO

Welcome to CHECKERS  ———————————————— Program continues, slowly
```

The Profiler counts the number of times each statement is encountered. This counting of each statement slows down program execution. For this reason, it may not always be possible to profile programs that operate in a time-critical environment.

The Profiler generates a performance outline when the program terminates. Termination occurs when your program reaches the logical end of the program or when the program detects a fatal error condition. A Control-C (^C) interrupts the program and generates a profile at that point. Entering Control-C (^C)'s twice aborts the generation of the profile.

The Profiler listing has the same two columns of numbers as the Debugger listing (one column numbers each line of the source program and the other gives the statement number of the first statement on each line), plus an extra column of numbers at the far left of the listing.

This leftmost column lists the number of times the statement on that line is executed. If more than one statement appears on the line, the count applies only to the first statement on the line. To obtain an accurate count of each statement in the program, you can run your source program through the PASMAT formatter supplied with Pascal–2. The PASMAT 'S' directive reformats the code so that no more than one statement appears on each line. (PASMAT is described in the Utilities Guide.)

If no number is printed in the leftmost column, then that particular statement was never executed. You can sometimes detect logic errors in your program by scanning the profile output to find sections of code or perhaps entire procedures that are never executed.

A summary of the program's execution, procedure by procedure, appears at the end of the profile listing. Procedures are listed in the order they appear in your source code. Three columns of information are displayed for each procedure, as follows:

Statements This column lists the number of statements that appear in the definition of the procedure.

Times Called This column shows how many times each procedure is called during program execution.

Statements Executed This column has two figures. The first is the number of statements executed in the procedure. For example, a procedure that contains 10 assignment statements and is called 5 times will show 50 statements executed in the **statements executed** column. This direct relationship is valid only for very simple procedures. In most procedures and functions, loops and other control structures cause the number of "statements executed" to be much larger (or smaller) than you may expect at first glance. The second figure in this column is the percentage of statements executed in this procedure as compared to the total number of statements executed in the program. The total number of procedures and statements and the total number of statements executed are printed at the bottom of the procedure execution summary.

The following example profile from CHECKR shows that 2.6 million statements were executed. (To save space, only the Procedure Execution Summary and relevant portions of the profile listing are shown here.) The Profiler listing shows that the program spent most of its time in only a few procedures. For example, the summary shows that 21 percent of the total statements executed were in the 15-statement procedure **Check**. However, **Check** was called 71,212 times, so that percentage does not seem too far out of line. More interesting is that almost half a million statements (17.63 percent) were executed in the procedure **Initialize**. This number seems excessive because the procedure does nothing more than initialize variables and tables each time a board position is analyzed and was only called 1348 times. We may have a problem here.

PROCEDURE EXECUTION SUMMARY

Procedure name	statements	times called	statements executed	
NEWNODE	15	1390	18070	0.69%
INITIALIZE	17	1348	459668	17.63%
SCAN	32	1348	120580	4.62%
CHECK	15	71212	567111	21.75%
ANALYZEMOVE	40	25362	516325	19.80%
ANALYZE	38	1348	298568	11.45%
UNPACKNODE	54	1348	60660	2.33%
PACKNODE	23	1348	22768	0.87%
SCOREGRADIENT	15	1348	250028	9.59%
SCOREBOARD	54	1348	99228	3.80%
EVALUATEBOARD	5	1348	6740	0.26%
DISPLAYBOARD	22	41	5453	0.21%
EXTRACT	18	715	7328	0.28%
KILL	11	1388	13621	0.52%
PRUNE	3	219	657	0.03%
INIT	165	1	1575	0.06%
COMPARE	14	4128	24768	0.95%
INSERT	26	1843	40490	1.55%
DUMPNODE	11	0	0	0.00%
GENMOVE	18	1273	17822	0.68%
GENJUMP	53	75	4389	0.17%
MOVEPIECE	12	1790	32100	1.23%
EXPAND	17	239	20372	0.78%
POSITIONCURSOR	2	0	0	0.00%
MAKEMOVE	55	306	7371	0.28%
DESCEND	26	197	3592	0.14%
FULLEXPAND	45	127	6046	0.23%
READMOVE	6	2	12	0.00%
DECODE	12	0	0	0.00%
READFILENAME	9	0	0	0.00%
GETUSERMOVE	108	1	90	0.00%
MAIN	91	1	2406	0.09%

There are 1032 statements in 32 procedures in this program.
2607836 statements were executed during the profile.

Because we suspect a problem in the procedure Initialize, we examine the profile output associated with that procedure. The first column of numbers is the statement execution count. The second column is the line number of the statement in the source file. The third column of numbers is the statement number of the statement. (This statement number is the same number used by the Debugger.)

The Profiler listing for procedure Initialize is:

```
        173          procedure Initialize;
        174            var
        175              I: integer;
 1348   176    1       begin { start of Initialize }
 1348   177    2         for I := - 5 to 49 do begin
74140   178    3           Vacant[I] := false;
74140   179    4           Friend[I] := false;
74140   180    5           Enemy[I] := false;
74140   181    6           FriendKing[I] := false;
74140   182    7           EnemyKing[I] := false;
74140   183    8           Protected[I] := false;
        184                end;
 1348   185    9         Pinned := 0;
 1348   186   10         Threatened := 0;
 1348   187   11         Umobil := 0;
 1348   188   12         Denied := 0;
 1348   189   13         BlackPieces := 0;
 1348   190   14         WhitePieces := 0;
 1348   191   15         Center := 0;
 1348   192   16         MoveSystem := 0;
 1348   193   17         EnemyHasKings := false;
        194              end; { of Initialize }
```

In statements 3 through 8, a for loop is initializing several Boolean arrays of the same type. Each assignment inside the loop is executed 74,140 times—a very inefficient way to initialize these arrays. Instead, we can modify the program to initialize one array, then assign that array to the other arrays to be initialized.

The effect of the modification is apparent in this new profile of the same section of code.

```
        173          procedure Initialize;
        174            var
        175              I: integer;
 1732   176    1       begin { start of Initialize }
 1732   177    2         for I := - 5 to 49 do begin
95260   178    3           Vacant[I] := false;
        179                end;
 1732   180    4         Friend := Vacant;
 1732   181    5         Enemy := Vacant;
 1732   182    6         FriendKing := Vacant;
 1732   183    7         EnemyKing := Vacant;
 1732   184    8         Protected := Vacant;
 1732   185    9         Pinned := 0;
 1732   186   10         Threatened := 0;
 1732   187   11         Umobil := 0;
 1732   188   12         Denied := 0;
 1732   189   13         BlackPieces := 0;
 1732   190   14         WhitePieces := 0;
 1732   191   15         Center := 0;
 1732   192   16         MoveSystem := 0;
 1732   193   17         EnemyHasKings := false;
        194              end; { of Initialize }
```

The result is clear: Instead of six assignments, each of which is executed 74,140 times, we have one assignment executed 95,260 times. (The execution numbers differ from the sample execution summary because the CHECKR program uses random numbers to play a different game each time it is run.) Overall, the Program Execution Summary shows that the time spent in the **Initialize** procedure has dropped from 17 percent to 4 percent of the total program. By rewriting six lines, we have improved performance by 11 percent.

Further, the number of times Statement 3 is executed can be reduced by the use of a global array initialized only once at the start of the program.

Similar optimizing techniques may be applied to other parts of the program. The Procedure Execution Summary indicates where the effort can best be applied—and where it cannot. For example, the program spent 21 percent of its time in the 15-statement procedure called **Check**. The trimming of even one statement from this procedure could significantly improve performance. On the other hand, one of the larger procedures in the CHECKR program is **Genjump**, containing 53 statements. The program, however, spent much less than 1 percent of its time in this procedure. Even by eliminating this procedure completely, we would improve program performance by only a trifling amount.

Two warnings: First, a statement count is not identical to "work." Complex statements take more time to execute than simple statements and this time is not measured. Second, the percentages shown in the **statements executed** column are percentages of execution counts, not execution time. For compute-bound programs such as CHECKR, the execution percentage closely approximates the percentage of time spent in the procedures. I/O-bound programs, however, may spend much of their execution time opening files or waiting for the disk to transfer information to memory. In this case, the execution count percentages may differ significantly from the real amount of time spent in the procedures.